An Axiomatic Value Model for Isabelle/UTP

Frank Zeyda[1], Simon Foster[2], and Leo Freitas[3]
f.zeyda@tees.ac.uk simon.foster@york.ac.uk leo.freitas@ncl.ac.uk

[1] Teesside University, School of Computing, Middlesbrough, TS1 3BA, UK.
[2] University of York, Department of Computer Science, York, YO10 5GH, UK.
[3] Newcastle University, School of Computing Science, Newcastle, NE1 7RU, UK.

Abstract. The Unifying Theories of Programming (UTP) is a mathematical framework to define, examine and link program semantics for a large variety of computational paradigms. Several mechanisations of the UTP in HOL theorem provers have been developed. All of them, however, succumb to a trade off in how they encode the value model of UTP theories. A deep and unified value model via a universal (data)type incurs restrictions on permissible value types and adds complexity; a value model directly instantiating HOL types for UTP values retains simplicity, but sacrifices expressiveness, since we lose the ability to compositionally reason about alphabets and theories. We here propose an alternative solution that axiomatises the value model and retains the advantages of both approaches. We carefully craft a definitional mechanism in the Isabelle/HOL prover that guarantees soundness.

1 Introduction

Much work has already been done in developing semantic models of particular programming languages and modelling notations. The Unifying Theories of Programming (UTP) [10] put forward an agenda of relating and combining such models in order to facilitate the development of sound foundations for highly-integrated languages that incorporate multiple paradigms, such as concurrency [16], object orientation [20], and time [21], to name a few only.

The importance of the UTP is to justify verification techniques that involve a heterogeneous set of notations, methods and tools. This is becoming an integral part of certification standards such as DO-178C in avionics [19], and motivated work in mechanising the UTP in theorem provers. Machine-checked proofs about the formalism(s) in use may thus become part of the certification evidence, in addition to verification proofs of actual systems and software components.

Several mechanisations of the UTP are currently available [17,25,6,3,7]. The majority of them uses HOL-based provers, namely ProofPower-Z [17,25] and Isabelle/HOL [6,7]. Only [3] develops a proof system and tool from scratch. The use of Isabelle/HOL in the aforementioned works is motivated by the high level of adaptability and automation afforded by this prover. This is, for instance, due to its ability to interface with external tools such as powerful SMT solvers [1].

Although Isabelle/HOL appears to be an attractive choice for a proof tool, its type system forces us into a compromise when encoding the binding and

predicate model of UTP theories. UTP theories are, in essence, characterised by subsets of predicates over some alphabet of variables. Predicates are typically encoded by sets of bindings, namely those that render the predicate true. Bindings associate alphabet variables with values. A fundamental part of any UTP reasoning framework is hence the representation of bindings and values.

Where the existing works on UTP mechanisation most notably differ is in how they encode the binding and value model of UTP theories. We here distinguish a deep and a shallow approach. In a deep approach as adopted by [14,17,25], a monomorphic value type with a fixed representation is introduced, typically as a datatype. This leads to a monomorphic binding model, and thereby, a monomorphic predicate model. It permits a high level of expressiveness by allowing us to define operators that inspect and modify the alphabets of predicates. A downside is that the value model must be *a priori* fixed and therefore cannot be extended. Moreover, certain constructions, such as arbitrary sets and functions, are difficult to support as they are not permissible in recursive datatype definitions.

In a shallow approach, as adopted by [5,6], the binding type is kept abstract by using a HOL type variable in place of it. This leads to a polymorphic (type-parametric) binding and predicate model. Therein, variables can only have an abstract representation, and we cannot prove properties about them until the binding model is (at least partially) instantiated — typically, using extensible record types to retain a degree of modularity. Doing so, however, forfeits the ability to compositionally reason about predicate alphabets. A crucial advantage of the shallow model is that UTP values can be drawn from *any* HOL type, and reasoning is much simplified since we are able to directly employ HOL theorems and tactics; the shallow model is moreover naturally extensible.

We here present an alternative and novel approach that uses an axiomatic value model. It combines the advantages of the deep and shallow approach, with no added complexity for the user. Our contribution here is not only relevant to mechanised proof support for the UTP, but indeed any kind of semantic language embedding in HOL. The choice of Isabelle/HOL is a pragmatic one: we benefit from an adaptable and open architecture, as well as powerful external proof tools that we can readily interface with. While dependently-typed logics and provers may tackle the issue we address in other ways, we nonetheless believe there is important scientific benefit in solving it in the context of HOL, too.

Our terminology of a deep and shallow approach ought not be confused with the terminology of a deep or shallow **embedding**. Whereas an embedding is classified as deep if it encodes the syntax of the embedded language, this paper is only concerned with the nature of semantic models. We remark that at the core, the UTP can in fact be viewed as a shallow embedding of program logic.

The structure of the paper is as follows. In Section 2, we review the UTP and Isabelle/HOL. Section 3 surveys the existing UTP mechanisations, comparing their encoding approaches for values and bindings in detail. Section 4 introduces our axiomatic value model in mathematical terms, and Section 5 describes its sound implementation in Isabelle/HOL. Lastly, in Section 6 we give an example of its use, and conclude and discuss future work in Section 7.

2 Preliminaries

In this section, we discuss preliminary material: the UTP in Section 2.1, and Isabelle/HOL in Section 2.2.

2.1 Unifying Theories of Programming

The Unifying Theories of Programming (UTP) [10] is a mathematical framework for describing and unifying the formal semantics of programming and modelling languages within the same descriptive environment of the alphabetised relational calculus. A UTP theory consists of an alphabet of variable names, a signature of language constructs, and a set of constraints (called healthiness conditions). Relations are encoded by alphabetised predicates: that is, predicates that contain additional information about the relation's alphabet.

Alphabets identify observational variables whose values are relevant to characterise system behaviour within a given paradigm. We use undecorated variables for initial, and dashed variables for intermediate or final observations. The alphabet of each theory contains variables relevant to the description of its programs, as well as auxiliary variables used to record aspects of the paradigm. For instance, the UTP theory of designs uses a boolean variable ok to record the program has started, and ok' to record that the program has terminated. The seminal book [10] on UTP is not precise on typing, but it is generally acknowledged that we operate in a typed language and logic setting, with common mathematical structures being available, such as sets, functions and sequences.

Through appropriate choice of variables and mathematical structures, it is possible to express the desired features of a programming notation in an elegant and concise way. The underlying UTP theory must select the appropriate and relevant subset of variables to represent intended behaviours. The signature of a theory is the language syntax, and the meaning of every program is given as a predicate restricted to the selected alphabet and signature.

Healthiness conditions formalise constraints on the semantic model: we only consider predicates that satisfy the healthiness conditions of a theory as valid models of computations within that theory. Importantly, healthiness conditions sometimes depend in their definition on the alphabet of the theory in which they reside. For instance, the theory of methods in [24] adds one constraint for each method variable m that is present in the theory's alphabet. This illustrates the nominal character of the UTP logic: variables are treated as first-class objects, with the αP operator yielding the alphabet of a predicate P as a set.

One can think of the UTP as a 'theory supermarket': whatever theoretical mechanisms are needed for a particular application, pick the appropriate UTP theories and link them to provide the laws and compositional refinement notion to verify specifications all the way down to code. The use of Galois connections is pervasive within UTP theories as a means to enable the description of formal links between a variety of paradigms, justifying the use of the same (formal) universe of discourse. In this utopian view of programming, the underlying mathematics are often challenging and profit from a mechanised reasoning

framework, where the customer of the theory supermarket can be assured that the ingredients she picks soundly combine when preparing her theory.

Having said that, when it comes to making use of such theories in an industrial setting, or on examples beyond the blackboard, a suitable arrangement of technical details is required in order to use proof assistants. That is, before we can focus on any proof obligations born from modelling, we first need to shape and polish models to fit the needs of a mechanical theorem prover. The most fundamental problem tackled in this paper is therefore the description of an extensible and (expressively) rich value model. We claim this is as much part of UTP theory engineering as defining operators and healthiness conditions.

Our key objective here is to free the language designer from any restrictions that may be imposed by the embedding of the UTP logic in a HOL theorem prover; that is, without having to compromise on expressivity elsewhere.

2.2 Isabelle/HOL

Isabelle/HOL [13] is a popular theorem prover for Higher-Order Logic (HOL). It follows the design of LCF [9] in protecting the user from unsound deductions: theorems can only be generated through valid inferences that, ultimately, rely on the consistency of a small logic kernel of axiomatic rules only.

The Isabelle framework itself is agnostic to the logic being used. There exist, for instance, instantiations of it for First-Order Logic and Zermelo-Fraenkel set theory. Isabelle provides natural-deduction-style proof rules and an underlying proof engine to conveniently perform backward and forward inferences. In addition, several powerful external provers can be easily invoked from within Isabelle. A structured proof language called ISAR is also part of the system.

Types in Isabelle/HOL can be defined in various ways. The most basic type declaration is via a **typedecl** $('t_1, 't_2, \ldots)\ T_{new}$, which introduces a new given type T_{new} without any constructor functions. The $'t_1$, $'t_2$, and so on, are possible parameters of the type. All we know about such types is that they are non-empty.

Type definitions are supported by way of:

$$\textbf{typedef}\ ('t_1, 't_2, \ldots)\ T_{new} = S :: ('t_1, 't_2, \ldots)\ T_{exists}\ \texttt{set}$$

where S is some (non-empty) subset of values of some existing type T_{exists} to which the newly-defined type is deemed to be isomorphic. We thus obtain a pair of abstraction and representation functions which are internally axiomatised to provide a bijection from S into the carrier set of T_{new}.

More sophisticated type definitions can be achieved with the **datatype** command for (co)inductive datatypes and **record** command to introduce extensible record types, although underneath the HOL system reformulates both in terms of plain **typedef**s. This definitional style of implementing high-level features guarantees that soundness is necessarily preserved.

Isabelle additionally supports type classes. They can be viewed as contractual specifications on types. A type may instantiate a particular type class C, and such can be formulated as a requirement $'\texttt{a} :: C$ on some type $'\texttt{a}$. We note that the symbol '::' in HOL is used for both, typing and type-class membership.

Name	Developers	Proof System
UTP in ProofPower-Z	Nuka [14], Oliviera [17], and Zeyda [25]	ProofPower-Z
Isabelle/*Circus*	Feliachi et al. [5,6]	Isabelle/HOL
Isabelle/UTP	Foster and Zeyda [7]	Isabelle/HOL
$U{\cdot}(TP)^2$ (Saoithín)	Butterfield [2,3]	Custom

Table 1. Existing works that mechanise the UTP framework.

3 UTP embedding approaches

In this section, we survey the existing mechanisations of the UTP and their approaches to encoding values, bindings and predicates. A complete list of current works is presented in Table 1. We note that there are three mechanisations that target ProofPower-Z, but they are very similar in how they encode the predicate and value model. All works except for $U{\cdot}(TP)^2$ are definitional, meaning that they extend HOL conservatively; this guarantees consistency of the embeddings. As $U{\cdot}(TP)^2$ uses its own logic, consistency must be argued by other means.

All HOL-based embeddings create some type $\mathcal{P}$ for alphabetised predicates, either as a type synonym — in some cases with associated constraints, or HOL type definition. The model of $\mathcal{P}$ is typically the set of bindings $\mathbb{P}(\mathcal{B})$ over some binding type $\mathcal{B}$. In all works except Isabelle/*Circus*, $\mathcal{P}$ also includes explicit information about the predicate's alphabet. We note that Isabelle/*Circus* represents predicates as characteristic functions $\mathcal{B} \Rightarrow \mathit{bool}$, but this does not limit generality of our discussion, as being equivalent to a set-based encoding.

3.1 A shallow predicate model

A shallow predicate model is adopted by Isabelle/*Circus* [5,6]. The binding notion is kept abstract, using a HOL type variable such as $'s$ for it. UTP variables are likewise modelled abstractly, by way of pairs consisting of a getter and update function. The types of these functions are recaptured below.

$$\mathit{get} :: 's \Rightarrow 'a \quad \text{and} \quad \mathit{update} :: 's \Rightarrow 'a \Rightarrow 's$$

Above, $'a$ determines the HOL type of the variable. The *get* function extracts the value of the variable from a binding, and the *update* function modifies the binding to assign a new value to the variable. Variables hence do not have a symbolic identity that is, for instance, formalised by an encoding of names.

A key advantage of this approach is that UTP variables can range over arbitrary HOL types $'a$; a downside is that we cannot prove anything about them unless the binding type $'s$ is concretised, so that the *get* and *update* functions may be concretely defined. In Isabelle/*Circus*, instantiation of the binding type accompanies UTP theory development. It is done partially and incrementally, by way of extensible records. For instance, to encode the UTP theory of designs,

we have to create a record type $(\!| ok :: bool, 'more |\!)$ to encode the variable ok. The type $'more$ here corresponds to the open extension of the record type and allows us to subsequently add further variables to that theory.

The use of extensible records retains a certain degree of modularity in defining generic connectives that apply to predicates with different alphabets. These connectives are typically encoded by operations on the binding sets. Unification of the binding types is therefore needed to apply these operators. As an example, we may unify the following binding types $(\!| ok :: bool, 'more |\!)$ and $(\!| ok :: bool, x :: nat |\!)$ by instantiating $'more$ with $(\!| x :: nat |\!)$. The first corresponds to the (extensible) design alphabet $\{ok :: bool, \ldots\}$, and the second to the closed design alphabet $\{ok :: bool, x :: nat\}$ including a program variable x.

A ramification of this approach is that each time we introduce a variable, we effectively have to create a host-logic record type for it. It is therefore non-compositional in the treatment of alphabets. Variables, despite their abstract representation, are not first-class citizens in this treatment: we cannot create them on-the-fly or collect them in sets.

In a shallow model, the value universe $\mathcal{U}$ may potentially include any Isabelle/HOL type. The binding type $\mathcal{B}$ is equated with open and closed record types; this makes the predicate type $\mathcal{P}$ parametric in the extension type of (open) records. New record types are created through Isabelle's declarative mechanism, ensuring soundness. In this approach, complexity arises as record types have to be created as UTP theory development unfolds; complexity is, however, alleviated by a thin layer between object and host-logic value models.

3.2 A deep predicate model

The ProofPower-Z works [14,17,25] use a deep predicate model by creating a fixed value universe $\mathcal{U}$ as an inductive datatype that supports the construction of various basic and composite values. Below, $\mathbb{F}(S)$ yields the finite subsets of S.

$$VALUE ::= Nat(\mathbb{N}) \mid Bool(\mathbb{B}) \mid Pair(VALUE \times VALUE) \mid Set(\mathbb{F}(VALUE)) \mid \ldots$$

This approach leads to a monomorphic predicate type $\mathcal{P}$, because bindings $\mathcal{B}$ can be equated with the function space $VAR \Rightarrow VALUE$, where both the domain and range types are monomorphic. UTP variables (type VAR) are encoded symbolically as strings, with some added information for dashes and subscripts. The work [25] adds to this a (monomorphic) model of types to formalise well-typed constructions. In that model, variables are encoded by name and type pairs.

In a deep predicate model, we are able to introspect and reason about the alphabets of predicates since variables are treated as first-class objects. This provides more expressivity to mechanise UTP theories, since functions can be formalised that manipulate predicates and their alphabets in any conceivable manner. We discuss an example where this is needed in Section 6.

A downside of the deep approach is that the value model is not extensible, since the $VALUE$ type (universe) must be defined upfront. The use of datatypes imposes further restrictions. For instance, we cannot support general set-valued

constructions as to avoid well-known inconsistencies [22], which is why the argument of $Set(\_)$ above must be a finite set. Recent advances in using categorical foundations for datatypes in Isabelle/HOL [23] have relaxed that restriction to furthermore permit infinite sets with bounded cardinalities, but this is still more restrictive than HOL sets in general.

The use of a deep model is often inevitable if we perform a deep embedding, since it enables us to formalise the mapping from syntax to semantics within the host logic. While a deep model offers more expressiveness at the level of predicates and UTP theories, it incurs restrictions with regards to what kind of values can be supported. Moreover, operators and theorems about (HOL) value types need to be 'lifted' into the unified $VALUE$ type, resulting in a larger number of definitions and underlying proof infrastructure to burden the user.

3.3 A hybrid predicate model

Isabelle/UTP [7] adopts a hybrid approach to alleviate some of the downsides of a deep predicate model while retaining its expressivity. Rather than using a polymorphic type $'s$ for bindings, it introduces an abstract type $'a$ for the values themselves. This type, unlike in Isabelle/*Circus*, does not need to be instantiated as the UTP theory hierarchy unfolds. Instead, we create type classes to inject particular desired HOL types into it. The type classes introduce the abstraction and representation function for the respective value. An example follows.

```
class INT_SORT =
  fixes MkInt :: "int ⇒ 'a::TYPED_VALUE"
  fixes DestInt :: "'a::TYPED_VALUE ⇒ int"
  assumes MkInt_inv : "DestInt(MkInt x) = x"
  assumes DestInt_inv : "y :u IntType ⟹ MkInt(DestInt y) = y"
```

The constant `IntType` and the operator $:_u$ are provided by the `TYPED_VALUE` type class, whose definition we omit for brevity. We can indeed think of the classes as type definitions that 'reuse' the target type to be defined. To prove consistency, we have to show that an aggregation of type classes (one for each value notion used by a UTP theory) can be instantiated. Logically, this corresponds to showing that the abstract value type has a model that satisfies the assumptions of all aggregated type classes. Yet in practice, such a proof has to be carried out for every UTP theory, based on what value notions are used by the theory.

With the above, we can formalise constraints on the value model of particular UTP theories through class constraints on $'a$. For instance, the theory of designs requires the presence of a value type to encode booleans for its auxiliary variables ok and ok', and this can be captured by a class constraint $'a :: \texttt{BOOL\_SORT}$ on all definitional entities that play a part in the encoding of that UTP theory.

In this approach, the universe $\mathcal{U}$ need not be fixed upfront. We can inject new types into it as we go along. To prove consistency, which now becomes a 'proof obligation' to be discharged by the user, we are, however, still restricted to value notions that have a model within HOL. We note that the hybrid approach can be 'abused' as an axiomatic treatment, for instance, to support general sets and

functions as UTP values but in doing so, we introduce the possibility of localised inconsistencies into the value model. This is not safe since the consistency issue then rests with the user rather than the mechanised framework.

In conclusion, it seems we cannot have our cake and eat it: none of the existing mechanised UTP systems gives us an unconstrained and provably-sound value model *and* an expressive (compositional) predicate model. In the remainder, we propose a new axiomatic approach that satisfies both desiderata.

4 An axiomatic value model

We next describe our value model in general mathematical terms. Section 4.1 examines the HOL universe, and Section 4.2 our axiomatic UTP universe.

4.1 The HOL universe

The standard set-theoretic semantics of HOL prescribes the von Neumann universe $V_{\omega+\omega} \setminus \{\varnothing\}$ (without the empty set) as a minimal model for its possible type constructions [18]. The von Neumann universe V_i is inductively defined for some index i by repeated application of the power-set for ordinal indices β, and generalised union for limit ordinals λ.

$$V_0 \mathrel{\widehat{=}} \varnothing \qquad V_{\beta+1} \mathrel{\widehat{=}} \mathbb{P}(V_\beta) \qquad V_\lambda \mathrel{\widehat{=}} \bigcup_{\beta<\lambda} V_\beta$$

Each limit ordinal index corresponds to the union of all sets constructed up to that level. In HOL, every finite type is representable by some V_n (for $n \in \mathbb{N}_{>0}$), and every infinite type by some $V_{\omega+n}$. For example, `nat` and `int` correspond to V_ω, and `real` and `nat set` correspond to $V_{\omega+1}$. In Isabelle/HOL specifically, types can be constructed either by composition of existing parametric types, or by definition of new types through identification of a suitable non-empty subset of some existing type [11,12]. The built-in types of Isabelle/HOL are:

- the boolean type `bool` containing the elements `True` and `False`;
- the infinite type `ind` whose cardinality is that of the naturals;
- the parametric function type $\sigma \Rightarrow \tau$ for HOL types σ and τ.

From these three types, all standard HOL types can be produced, including the power type σ `set` ($\mathrel{\widehat{=}}\ \sigma \Rightarrow$ `bool`), the product type $\sigma \times \tau$, and the sum type $\sigma + \tau$. The constructions are performed via the **typedef** command, though the σ `set` type is technically axiomatised in Isabelle/HOL. This, however, is merely for convenience — its definition as a function type is equally feasible. It shows though, in defence of our solution, that Isabelle/HOL itself does not shy away from axiomatisations where we can provide strong evidence for consistency.

We conclude that all types in Isabelle/HOL of higher cardinality than $|\mathbb{N}|$ must be constructed by a (finite) repeated application of the power-type constructor σ `set`, with their cardinality being bounded by $V_{\omega+n}$ for some $n \in \mathbb{N}$. Thus it is impossible to define a type as large as $V_{\omega+\omega}$ within HOL itself, when using only the standard mechanisms for type definition.

The above implies that it is not possible to define a universal type $\mathcal{U}$ in HOL into which all HOL types are injectable. The existence of such a type in HOL would moreover lead to inconsistency, since there would then have to exist an injection $\mathcal{U}$ `set` into $\mathcal{U}$ itself, which Cantor's theorem forbids. In introducing $\mathcal{U}$ axiomatically, namely for UTP value and type models, it is, in essence, the latter that we have to protect ourselves from.

There have been several attempts to formalise a larger universe in HOL than the standard definitional mechanisms allow. HOL-ST [8] is an experimental combination of HOL and set theory that axiomatises a universe consisting of ZFC set constructions. HOL-ST was later adapted to create Isabelle/HOLZF [15] which axiomatises the ZFC universe as a type `ZF` alongside other HOL types; the motivation of that work was to formalise the notion of *Partisan Games* [15] as they cannot be captured through permissible **datatype** constructions in HOL.

Our approach here has the same aim as HOL-ST in using an **axiomatization** to provide a type that is 'larger' than any type definable in HOL, but unlike HOL-ST we want to make it possible to directly inject existing HOL types in our new (axiomatic) type. For this, it is sufficient to declare a type UVal and postulate three axioms that provide injectivity and type reflection from HOL into UVal. The next section discusses the axiomatisation in general terms.

4.2 The UTP universe

In this section, we give a semi-formal exposition of our axiomatic UTP universe, which will be formally mechanised in Section 5. We presuppose the existence of a class Type of HOL types, and also a universe HOL of HOL values. We recall that the latter cannot be defined in HOL as a set, and we therefore refer to it here as a (proper) class. For simplicity, we do not directly consider polymorphism and treat each type $\sigma \in \mathsf{Type}$ as a fully-instantiated monomorphic type. Hence, no two types can possess a common element. Our objectives are:

- The creation of a universe type UVal into which the values of a suitable subset of permissible HOL types can be soundly injected;
- reflection of the HOL typing relation $v :: t$ into UVal, also allowing us to explicitly reason about typing within UVal.

Our universe will be implemented through monomorphic types. This enables us to form definitions and theorems that effectively quantify over HOL types. Each objective is characterised by an additional axiom that we will describe. These axioms are conceptual, and do not correspond precisely to the Isabelle axioms which cannot, for instance, have typing statements like $x :: \sigma$ as caveats or talk explicitly about the HOL universe of values. The axioms will therefore require some refinement before their implementation into Isabelle/HOL, which we describe in Section 5. We will also prove some necessary theorems implied by these axioms, which our implementation satisfies.

Our UTP universe is characterised by a declared Isabelle type UVal, together with a polymorphic injection function $\mathit{InjU} : \mathsf{HOL} \Rightarrow \mathsf{UVal}$, a projection function

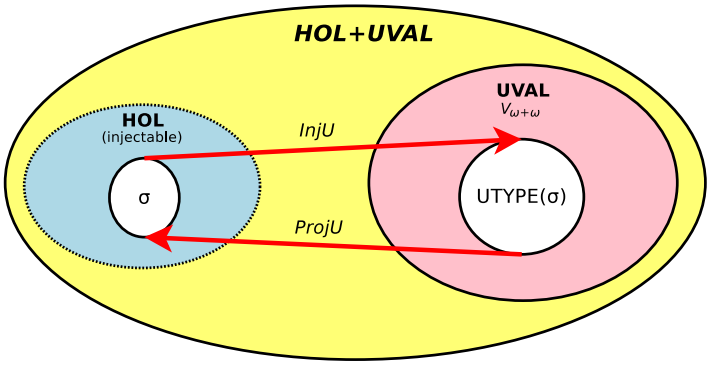

Fig. 1. Relation between the HOL and UVal universes.

ProjU : UVal ⇒ HOL, and a type mapping *UTYPE* : Type ⇒ UType. The application $UTYPE(\sigma)$ encodes a HOL type σ as a suitable value of a monomorphic type UType that represents HOL types. We also have a reflected typing relation $x :_u t$, for $x \in$ UVal and $t \in$ UType. We visualise the behaviour of these functions in Figure 1. Every HOL type $\sigma \in$ Type can be injected into a corresponding subset of UVal, by application of *InjU*. Moreover, all values within UVal can be projected back to their corresponding HOL type.

We now formally specify the behaviour of these functions through three axioms that augment the axioms of HOL:

1. AxValBij. For any $\sigma \in$ Type, *InjU* is a bijection between the values of σ and those of $UTYPE(\sigma)$, with *ProjU* being its inverse.
2. AxTypeRefl. The reflected typing relation is sound and complete with respect to HOL typing, such that $InjU(x) :_u UTYPE(\sigma)$ if and only if $x :: \sigma$.
3. AxTypeNonempty. For any $t \in$ UType, there exists a value $v \in$ UVal such that $v :_u t$.

Axiom AxValBij indirectly ensures that the cardinality of any HOL type is less or equal than that of UVal, as stated by the following theorem.

Theorem 1. *For any $\sigma \in$ Type, the cardinality of σ is no greater than that of* UVal*, that is $|\sigma| \leq |\mathsf{UVal}|$.*

Proof. *InjU*, by AxValBij, is an injection from σ to UVal. This is sufficient to demonstrate the required cardinality relationship. □

Furthermore, we can show that UVal has a strictly greater cardinality than any HOL type.

Theorem 2. *The cardinality of any HOL type $\sigma \in$ Type is strictly less than the cardinality of* UVal*, that is $|\sigma| < |\mathsf{UVal}|$.*

Proof. We prove this by contradiction. Assume that $|\sigma| \geq |\mathsf{UVal}|$, then either $|\sigma| > |\mathsf{UVal}|$ or $|\sigma| = |\mathsf{UVal}|$.

– If $|\sigma| > |\mathsf{UVal}|$, we obtain a contradiction by Theorem 1.
– If $|\sigma| = |\mathsf{UVal}|$, from $\sigma \in \mathsf{Type}$ we also have that $\sigma\ \mathtt{set} \in \mathsf{Type}$. By Cantor's theorem we have that $|\sigma\ \mathtt{set}| > |\sigma|$, and hence $|\sigma\ \mathtt{set}| > |\mathsf{UVal}|$. Again, by Theorem 1 this leads to a contradiction. □

A corollary of Theorem 2 is that neither UVal nor any type with a cardinality equal to or greater than UVal can be a HOL type.

Corollary 1. $\forall\, t \bullet |\mathsf{UVal}| \leq |t| \Rightarrow t \notin \mathsf{Type}$

Proof. By Theorem 2 we have $|t| < |\mathsf{UVal}|$, and by transitivity of $<$ thus follows the contradiction $|\mathsf{UVal}| < |\mathsf{UVal}|$. □

It is therefore essential to ensure that UVal cannot be made an element of Type for our logic to remain consistent. We can also demonstrate a number of necessary consequences of the type reflection axiom AxTypeRefl. Firstly, we require that each reflected type identify a unique HOL type.

Theorem 3. *The type mapping function UTYPE is injective for* $\sigma, \tau \in \mathsf{Type}$. *That is,* $UTYPE(\sigma) = UTYPE(\tau)$ *implies that* $\sigma = \tau$ *for all* $\sigma, \tau \in \mathsf{Type}$.

Proof. Assume $UTYPE(\sigma) = UTYPE(\tau)$ for $\sigma, \tau \in Type$. By non-emptiness of σ, there exists some value x with $x :: \sigma$. Thus $InjU(x) :_u UTYPE(\sigma)$ by axiom AxTypeRefl, and $InjU(x) :_u UTYPE(\tau)$ since $UTYPE(\sigma) = UTYPE(\tau)$. Converse application of AxTypeRefl finally yields $x :: \tau$. Because of disjointness of types in HOL, it follows that $\sigma = \tau$. □

Note that we cannot show from the axioms that all reflected types possess a witness. Namely, that for any $t \in \mathsf{UType}$, there exists a value $x \in \mathsf{UVal}$ such that $x :_u t$. To show this, a sufficient condition is that every element in UType is the image of some permissible HOL type $\sigma \in \mathsf{Type}$. In practice, this turns out to be too strong since, clearly, not all HOL types are permissible. The third axiom AxTypeNonempty thus guarantees non-emptiness of all reflected types.

For those types that are not in the image of $UTYPE$, non-emptiness is all that we know about their values. For other types, which *are* in the image of $UTYPE$, the axiom does not add any new knowledge, since for those types we can already prove from the axioms AxValBij and AxTypeRefl that they are non-empty. Hence this additional axiom does not pose a risk to consistency.

5 Implementation in Isabelle/HOL

In this section, we describe our implementation in Isabelle/HOL of the axiomatic value model that was proposed in the previous section.

5.1 UTP values and types

UTP Types Our goal is to associate UTP model types directly with (a subset of) the HOL types. HOL, in general, is not expressive enough to treat HOL types as values. However, the type-class mechanism is used in Isabelle/HOL to define an operator `TYPEREP('a)` that converts a HOL type `'a` into a representation of that type as a HOL value. The representation is in terms of a datatype `typerep`,

which is part of the standard HOL library and recaptured below.

```
datatype typerep = Typerep String.literal "typerep list"
```

It has a single constructor `Typerep` that takes both a string literal for the type's name, and a list of `typerep` objects corresponding to the arguments of a parametric type. The datatype encodes the structure of any monomorphic HOL type as a value, and is generally used as a limited facility to support reasoning about types in HOL. We effectively reuse it here to encode UTP model types, and for uniformity introduce a syntax abbreviation `utype` for it.

In order to apply the `TYPEREP('a)` operator to some type `'a`, the type `'a` must instantiate the type class `typerep` that defines how `'a` is to be represented. That type class is typically instantiated automatically by Isabelle when new types are created with **typedef**. We may hence reasonably assume that all HOL types we like to use in UTP theories instantiate `typerep`.

We proceed by introducing a polymorphic typing operator. We note that an implicit default sort constraint was placed on `'a` to be of class `typerep`.

```
definition p_type_rel :: "'a ⇒ utype ⇒ bool" (infix ":" 50) where
"x : t ⟷ TYPEREP('a) = t"
```

Above, `x : t` holds if the (HOL) value `x` is of UTP model type `t`. For instance, we can prove `(1 :: nat) : TYPEREP(nat)` but not `1 : TYPEREP(nat)` since numbers in HOL are polymorphic objects. This means that the type of `1` corresponds to some type variable `'a` of sort `typerep`. For such types, `TYPEREP('a)` cannot be simplified but we can still perform reasoning using unification. For this reason, our model in fact supports polymorphic types.

To facilitate proofs about typing, we provide a theorem attribute `typing` that collects all relevant theorems about typing, including the definitional theorem of `p_type_rel`. Simplification with added `typing` theorems typically discharges any kind of type conjecture, or otherwise reduces it to *false*. We implemented a hook into Isabelle/HOL's type definition packages that automatically collects the required theorems. This kind of proof engineering plays a crucial part in theory usability and proof automation, and is often overlooked in mechanisations.

We next examine the UTP value model. This is the core contribution of the novel mechanisation of the UTP in Isabelle/HOL that we developed.

UTP Values In agreement with both Section 4.2 and the earlier ProofPower-Z works, we introduce a monomorphic type `uval` for our UTP value model. We thus are able to retain all of the expressiveness of a deep binding and predicate model as in the works [17,25,7]. However, rather than giving `uval` a concrete definition, for instance, by virtue of a **datatype**, we leave it uninterpreted.

```
typedecl uval
```

In languages like Z, the above corresponds to the definition of a given type. As explained in Section 2.2, such types are not equipped with an abstraction or representation function. All we know about them is that they are non-empty.

Construction, destruction and typing of values in `uval` are axiomatised by three polymorphic functions: `InjU`, `ProjU` and `utype_rel`. For the third, we intro-

duce the infix notation `v` $:_u$ `t`. The following **axiomatization** introduces these constants as well as their defining axioms. This formalises our earlier axioms in Section 4.2 and is all that is needed to reason about UTP model values.

```
axiomatization
— Universal abstraction, representation and model typing relation.
  InjU :: "'a::injectable ⇒ uval" and
  ProjU :: "uval ⇒ 'a::injectable" and
  utype_rel :: "uval ⇒ utype ⇒ bool" (infix ":u" 50) where
— Axioms that determine the semantics of the above functions.
  InjU_inverse: "ProjU (InjU x) = x" and
  ProjU_inverse: "y :u TYPEREP('a) ⟹ InjU (ProjU y) = y" and
  utype_rel_def: "(InjU x) :u t ⟷ x : t" and
  utypes_non_empty: "∃y. y :u t"
```

The axioms have similarities with the standard axioms for type definitions [13]. First, we have a pair of injection theorems: `InjU_inverse` and `ProjU_inverse`. The first one is for the abstraction function (`InjU`), and the second one for the representation function (`ProjU`). An important difference to HOL type definitions is, however, that we do not merely inject the values of a single existing HOL type into the new type, but a universe of the values belonging to a collection of HOL types (HOL in Figure 1). That universe is identified by the type class `injectable`, whose purpose is explained later on in Section 5.2. It usually includes values of infinitely many HOL types because of type parametricity.

Since we here inject the entire carrier (`UNIV`) of a HOL type `'a`, contrary to **typedef**s there is no caveat present in the `InjU_inverse` injection theorem. Both injection theorems together implement the axiom AxValBij in Section 4.2. The sort constraint `'a::injectable` in the definition of the constants `InjU` and `ProjU` ensures that we cannot write any term `InjU x` where the argument `x` is not an injectable HOL type — Isabelle/HOL otherwise flags a type error. Likewise, the result of `ProjU` must always be chosen as to have an injectable type. The caveat of `ProjU_inverse` moreover ensures that the value we are projecting out of the UTP model and back into HOL has the correct type for the projection to be valid. Model typing `x` $:_u$ `t` is formalised by lifting polymorphic typing into `uval`. Our third axiom `utype_rel_def` hence corresponds to AxTypeRefl and ensures completeness and soundness of the reflective typing relation.

The fourth axiom `utypes_non_empty` encodes AxTypeNonempty, capturing that all UTP model types are non-empty. If all `utype` elements corresponded to injectable HOL types, this would follow automatically. However, since there are some HOL types that are inherently not injectable, the axiom requires that even those types have at least one value, though we do not know anything else about such types. The need for the axiom is technical: we want to ensure that there is a well-typed 'total' binding whose variables must range over *any* HOL type.

The axiomatisation gives us the ability to control what HOL types we like to inject into the UTP value model. This is crucial as the injection of certain types can lead to inconsistencies. We next discuss this issue and explain how we ensure that unsoundness cannot emerge from inappropriate use of our axioms.

5.2 Controlling injectability

The quintessential example that leads to inconsistency is injecting `uval` itself into the value model. Depending on the injection of other HOL types, in particular `'a set`, we are then able to derive a contradiction. Since `InjU` of (injectable!) type (`uval set`) ⇒ `uval` cannot be injective due to Cantor's theorem, the axiom `InjU_inverse` above clearly is violated in that case.

We could naively have implemented a mechanism that prevents the user from instantiating `uval` as `injectable` but this is not enough: a clever user might circumvent that mechanism by defining a new HOL type (via a **typedef**) that is equipotent to `uval` or even larger, and then the same problem arises if that new type is made permissible for injection into `uval`.

To solve this problem in a universal and robust manner, we first mechanise a notion of type dependency. We recall a type definition generally has the form:

typedef (`'a`, `'b`, ...) `new_type` = S :: (`'a`, `'b`, ...) T `set`

where the type term (`'a`, `'b`, ...) T only involves currently existing HOL types and S is a non-empty subset of the values of (`'a`, `'b`, ...) T. We observe that (`'a`, `'b`, ...) `new_type` depends on the types occurring in T and the type variables `'a`, `'b`, and so on. We formalise this dependency via a new type class `typedep`.

```
class typedep = typerep +
  fixes typedep :: "'a itself ⇒ typerep set"
```

This class extends Isabelle/HOL's existing class `typerep`. Any HOL type that instantiates it must additionally provide a function `typedep` that, given an element of `'a itself`, yields a set of type representations of HOL types that `'a` depends upon. The type constructor `'a itself` is primitive and conventionally used when a function is *polymorphically* parametrised by a HOL type. Polymorphism is crucial here since it determines resolution of `typedep` if applied to a particular HOL type. To simplify the application of `typedep`, we introduce a syntactic sugar that allows us to write `TYPEDEP(`T`)` for some HOL type T, instead of having to construct a corresponding value from `'a itself` and then apply `typedep` to it. Examples are `TYPEDEP(nat)`, `TYPEDEP(nat set)` and `TYPEDEP('a set)`.

A subtle issue is how we ensure that the class `typedep` is instantiated correctly. Below we give an example of instantiating `typedep` for the function type.

```
instantiation "fun" :: (typedep, typedep) typedep
begin
definition typedep_fun :: "('a ⇒ 'b) itself ⇒ typerep set" where
"typedep_fun t = TYPEDEP('a) ∪ TYPEDEP('b)"
instance by (intro_classes)
end
```

We first observe that the definition of `typedep` for the function type `'a ⇒ 'b` involves the recursive application of `typedep` (via the `TYPEDEP(_)` syntax) to the type parameters `'a` and `'b`, making precise that `'a ⇒ 'b` depends on `'a` and `'b`. We secondly observe that a type representation of the function type does

not itself occur in the right-hand side, namely there is no term such as $\ldots \cup \{\texttt{TYPEREP('a} \Rightarrow \texttt{'b)}\}$ included. The reason for this is that we are only interested in dependency to *ground types*, namely those types that are not defined in terms of other types and thus form the roots of the dependency hierarchy. This also ensures efficient evaluation of `TYPEDEP(_)` as resulting terms may become large.

There are indeed only two genuine ground types in HOL: `bool` and `ind`. Also, any type declaration via a **typedecl** construct introduces a new ground type. Therefore, `uval`, in our formalisation, crucially becomes a ground type, too. Although HOL's set type (`'a set`) and function type (`'a` $\Rightarrow$ `'b`) are not introduced by a type definition, we do not consider them as ground types.

For a type definition, such as the one on page 14, we would need to perform the following instantiation:

```
instantiation new_type :: (typedep, typedep, ...) typedep
begin
definition typedep_new_type ::
  "('a, 'b, ...) new_type itself ⇒ typerep set" where
"typedep_new_type t = TYPEDEP(T)"
instance by (intro_classes)
end
```

We observe that the dependency of a new type (`'a`, `'b`, ...) `new_type` is defined in terms of the dependency of its model type (`'a`, `'b`, ...) T. While the instantiation is uniform and easy to perform, it would constitute a risk to rely on the user to perform it. Instead, we implemented a hook in Isabelle/HOL that executes such instantiations automatically and outside the control of the user for each new type defined via a type definition. Isabelle/HOL provides an interface that allows one to execute such hooks (see the `Typedef.interpretation` ML function within the HOL source code). It, fortunately, even does so retrospectively for existing types. This again means that the user — just like with `typerep` — does not have to be concerned with the instantiation of `typedep` and precludes any unsoundness potentially arising from wrongly instantiating that class. For convenience, we lastly make `typedep` the **default_sort** for free type variables.

We are now in a position to define the `injectable` class in a safe manner. This class, we endow with two assumptions that have to be discharged upon instantiation of any HOL type as `injectable`.

```
class injectable = typedep + order +
  assumes utype_is_not_uval : "TYPEREP('a) ≠ TYPEREP(uval)"
  assumes utype_not_dep_uval : "TYPEREP(uval) ∉ TYPEDEP('a)"
```

The first assumption captures that the type we inject must not be the same as `uval`. The second uses the type-dependency mechanism by formalising that `uval` must not be in the set of types on which the type we inject depends. If both proof obligations can be discharged, we have established that injecting T into the UTP value model `uval` is safe and sound.

To facilitate the instantiation of HOL types as injectable, we provide an Isabelle command `inject_type` that discharges the above assumptions automatically. We note that this is for convenience and not for safety reasons — manual

instantiation means that the proof obligations would still need to be discharged. Their proof is usually not difficult and can be done by rewriting and automatic reasoning. Again, to facilitate proofs, we introduce an attribute to record theorems that are relevant to reason about type dependency. They are automatically collected when new types are defined and the class `typedep` is instantiated.

By default, we inject a useful subset of existing HOL types into the UTP value model, including `unit`, `bool`, `nat`, `int`, `char`, `real`, `fun`, `set`, `list`, `prod`, `sum` and `option`. We can, however, inject any custom type definition or datatype in exactly the same manner, as illustrated in the next section. While our implementation requires, to a certain degree, low-level ML programming of the proof system, all of this is done outside the Isabelle/HOL kernel and code — we did not have to change the prover's source distribution in any way. We also implemented useful error reporting to the user when a type cannot be injected as failing the caveats of the `injectable` class. Lastly, we note that mutually-recursive datatypes are implicitly supported since Isabelle/HOL endows such types with a common model, so that the recursive type dependency disappears when the underlying (non-recursive) **typedef**s are constructed under-the-hood by Isabelle/HOL.

We claim that our implementation is LCF-sound: this means that incorrect use of our tool cannot result in inconsistency of the logic. We deconstruct the evidence for this through the following reasoning chain.

1. We consider the approach to be 'mathematically sound', as a consequence of restricting injectable types to those that do not depend on `uval` (Section 4);
2. In the mechanisation, we use a type class to restrict injection to permissible values only, which excludes constructs that attempt to inject invalid types *already at the level of HOL type analysis* (Section 5.1);
3. The injectability caveat is formalised and enforced by endowing the above type class with two assumptions (proof obligations) (Section 5.2);
4. The proof obligations rely on the correct instantiation of `typerep` and `typedep` classes, but both instantiations reside outside the control of the user.

Our tool is, thus, not only an Isabelle/UTP extension to enable richer UTP value models, but also a low-level Isabelle/HOL language extension. A final point to note is that `injectable` in our design also imports the type class `order`, since we assume that any UTP value model is equipped with an order. This opens up further possibilities to mechanise High-Order UTP, which adds support for higher-order programming to UTP. The reason for this is that HO UTP relies on order relations on values, namely to (re)define common UTP operators such as skip, assignment and variable blocks in this context (Chapter 9 of [10]).

6 Example: mechanising a theory of object orientation

As an example, we consider Santos' UTP theory of object orientation [20]. In what follows, we illustrate how the axiomatic value model enables us to easily encode that theory, using our tool. The Isabelle 2015 sources and a report are available from `https://www.scm.tees.ac.uk/users/f.zeyda/utp2016/`.

The UTP theory of object orientation is an extension of the UTP theory of designs, and, therefore, includes the auxiliary boolean variables ok and ok' to record termination. Besides, it also includes additional auxiliary variables to capture specific aspects of the object-oriented paradigm. These variables and their types are explained below.

- cls of type $\mathbb{P}(CName)$ to record the names of classes used in the program;
- $atts$ of type $CName \nrightarrow (AName \nrightarrow Type)$ to record class attributes;
- sc of type $CName \nrightarrow CName$ to record the subclass hierarchy;
- an open set $\{\overline{\overline{m}}_1, \overline{\overline{m}}_2, \ldots\}$ of procedure variables for method definitions;
- an open set $\{\overline{m}_1, \overline{m}_2, \ldots\}$ of procedure variables for method calls.

Above, $CName$ is the set of all class names, $AName$ is the set of all attribute names, and $Type$ is defined as $CName \cup prim$ where the elements in $prim$ represent primitive types, like the integers or booleans. The functions $atts$ and sc are partial ($\nrightarrow$) since they only consider classes that are currently declared, namely those in cls. The function sc maps each class to its immediate superclass; the subclass relation is obtained via its reflexive and transitive closure: $C_{sub} \preceq C_{super} =_{\text{def}} (C_{sub}, C_{super}) \in sc^*$. There also exists a special class **Object** $\in CName$ that does not have a superclass.

The above description, which was taken from the literature, indeed gives us a very clear idea of how to design the value encoding for that theory. In doing so, however, we do not want to be constrained by a mechanised framework. The axiomatic value model lets us work at the level of HOL, using its definitional features as needed. Below we introduce the necessary types.

```
datatype cname = Object | Class "string"
datatype aname = Attr "string"
datatype prim = int | bool
datatype atype = PType "prim" | CType "cname"
```

Above, `cname` encodes $CName$, `aname` encodes $AName$, `prim` encodes $prim$, and `atype` encodes $Type$. The next step is to inject these types into the universal value type `uval`. As explained in the previous section, this is easily done with the following set of commands.

```
inject_type cname
inject_type aname
inject_type prim
inject_type atype
```

Behind the scene, the implementation of the **inject_type** command discharges the proof obligations that establish that the injections are sound. Here, this is the case since `uval` does not occur in the above **datatype** definitions.

It is worth noting that in order to support injection of datatypes into `uval`, we did not have to interface in any way with Isabelle's datatype package. This is because, ultimately, the definitional implementation of datatypes implies that everything boils down to plain type definitions, and our tool can readily handle those. For the same reason, **record** types are also supported out-of-the-box, as well as any other custom types that are definitional, which is the norm.

Name	Invariant	Description
OO1	$\mathbf{Object} \in cls$	**Object** is always a class of the program
OO2	$\mathrm{dom}\, sc = cls \setminus \mathbf{Object}$	Every class except **Object** has a superclass
OO3	$\forall C : \mathrm{dom}\, sc \bullet (C, Object) \in sc^+$	**Object** is at the top of the class hierarchy
OO4	$\mathrm{dom}\, atts = cls$	Attributes are defined for all classes
OO5	$\forall C_1, C_2 : \mathrm{dom}\, atts \mid C_1 \neq C_2 \bullet \mathrm{dom}(atts(C_1)) \cap \mathrm{dom}(atts(C_2)) = \varnothing$	Attribute names are unique across classes
OO6	$\mathrm{ran}(\bigcup \mathrm{ran}\, atts) \subseteq prim \cup cls$	Attributes have primitive or class types

Table 2. Healthiness conditions for the theory of object orientation.

Healthiness conditions The theory has seven healthiness conditions. They are characterised by invariants that constrain the permissible values of *cls*, *atts* and *sc*, as well as the procedure variables for methods. Table 2 summarises the first six constraints, which are related to *cls*, *atts* and *sc*. Intuitively, the invariant **OO1** requires **Object** always to be a valid class of the program. **OO2** and **OO3** determine the shape of the subclass relation: it has to be a tree with **Object** at its root. Attributes have to be defined for all classes (**OO4**), they have to be unique (**OO5**), and their types, if they are not primitive, must refer to declared classes (**OO6**). A further healthiness condition (**OO7**) not in Table 2 is inherited from the UTP theory of methods in [24]. Its shape is given below, where the function $\mathbf{SIH}(\_)$ is part of the UTP theory of invariants [4] and performs the conversion of invariants into design predicates over before and after states.

$$\mathbf{OO7}(P) = \mathbf{SIH}(\forall\, \overline{m}\ \overline{\overline{m}} \mid \{\overline{m}, \overline{\overline{m}}\} \subseteq \alpha P \bullet [\forall\, args \bullet \overline{m}(args) \Leftrightarrow \overline{\overline{m}}(args)]_0)(P)$$

This healthiness condition establishes a correspondence between procedure variables that are used for definition (double overbar) and call (single overbar) of methods. The purpose of **OO7**(P) is beyond the technical scope of this paper; we, however, observe that the quantifier above ranges over variables $\overline{m}$ and $\overline{\overline{m}}$ within the alphabet of predicate P. Encoding this condition may not be possible in a shallow model that does not allow us to quantify over alphabets.

We lastly present an example that illustrates how we encode the healthiness conditions. While a deep approach is non-negotiable in this case, the axiomatic value model enables us to express everything in terms of HOL concepts. This is done by 'lifting' HOL predicates into deeply-encoded UTP predicates. The lifting is performed by a simple rewrite engine that we implemented as part of the tool. With it, we may, for example, encode **OO5** as follows.

```
definition OO5 :: "upred" where
"OO5 = (∀ C1 ∈ dom atts .
        ∀ C2 ∈ dom atts . C1 ≠ C2 |
          dom (atts⋅C1) ∩ dom (atts⋅C2) = {})ₚ"
```

The tool that performs the lifting is invoked via the $(\_)_p$ construct. Inside the brackets, we may write plain HOL. The beauty of this is that we do not have to

be concerned with redefining any of the HOL operators that are used, such as $\in$, $\cap$, dom, and so on, for our value model, and neither recast laws and tactics for proof support. Our approach enables the development of a generic rewrite tool that circumvents all of this so that the user is able to work exclusively in HOL; the underlying details of the deep encoding are by and large concealed.

There are some useful aspects of the implementation that we did not discuss. For instance, we also provide a mechanism for parsing and rewriting HOL variables into UTP variables, in a way that we can take advantage of type-checking and unification. Our system is flexible: we can always escape the parser to combine unprocessed HOL with lifted predicate terms.

7 Conclusion

We have presented a novel approach to axiomatically encode value models of language embeddings. While we applied our work to the problem of mechanising the UTP framework, it remains applicable to *any* deep language embedding. The problem we addressed is to relax common restrictions on deep value models in HOL to support, for instance, general sets and functions. Our key contribution is the design of a solution and tool in Isabelle/HOL that is definitionally sound.

Beyond this, we put forward an approach to UTP theory engineering that enables and advocates working at the level of HOL rather than the formalised concepts and idioms of a particular mechanised framework. We claim that this is the crux in attracting academics to use a mechanised framework or theorem prover for the UTP, as we cannot expect users to acquire detailed knowledge of a mechanised framework or the nitty-gritty of a proof system. We hope that this work will set the future direction for UTP proof support, but accept that there is a price to pay in the currency of axioms for having our cake and eating it!

Future work will extend our mechanisation to be competitive with the currently available systems Isabelle/*Circus* [6] and Isabelle/UTP [7] in terms of the number of laws and mechanised theories. This work is mostly clerical and should not take a lot of time and effort. A second future work will isolate those parts that are independent of the UTP and only concerned with the value model, and publish this separately for the Isabelle/HOL community as a stand-alone tool.

Acknowledgement We would like to thank the anonymous reviewers for their helpful suggestions and conscientious reading of the paper.

References

1. J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT Solvers. In *Automated Deduction – CADE-23: 23rd International Conference on Automated Deduction*, volume 6803 of *LNCS*, pages 116–130. Springer, 2011.
2. A. Butterfield. Saoithín: A Theorem Prover for UTP. In *Proceedings of UTP 2010*, volume 6445 of *LNCS*, pages 137–156. Springer, November 2010.
3. A. Butterfield. The Logic of $U{\cdot}(TP)^2$. In *Proceedings of UTP 2012*, volume 7681 of *LNCS*, pages 124–143. Springer, August 2012.

4. A. Cavalcanti, A. Wellings, and J. Woodcock. The Safety-Critical Java memory model formalised. *Formal Aspects of Computing*, 25(1):37–57, January 2013.
5. A. Feliachi, M.-C. Gaudel, and B. Wolff. Unifying Theories in Isabelle/HOL. In *Proceedings of UTP 2010*, volume 6445 of *LNCS*, pages 188–206. Springer, November 2010. Tool url: http://afp.sourceforge.net/entries/Circus.shtml.
6. A. Feliachi, M.-C. Gaudel, and B. Wolff. Isabelle/*Circus*: A Process Specification and Verification Environment. In *Proceedings of VSTTE 2012*, volume 7152 of *LNCS*, pages 243–260. Springer, January 2012.
7. S. Foster, F. Zeyda, and J. Woodcock. Isabelle/UTP: A Mechanised Theory Engineering Framework. In *Proceedings of UTP 2014*, volume 8963 of *LNCS*, pages 21–41. Springer, May 2014.
8. M. Gordon. Set Theory, Higher Order Logic or Both? In *Proceedings of TPHOLs 1996*, volume 1125 of *LNCS*, pages 191–201. Springer, August 1996.
9. M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer, 1979.
10. T. Hoare and H. Jifeng. *Unifying Theories of Programming*. Prentice Hall Series in Computer Science. Prentice Hall, Upper Saddle River, NJ, USA, 1998. Freely available at http://www.unifyingtheories.org/.
11. M. Iancu and F. Rabe. Formalising foundations of mathematics. *Mathematical Structures in Computer Science*, 21(Special Issue 04):883–911, August 2011.
12. O. Kunčar and A. Popescu. A Consistent Foundation for Isabelle/HOL. In *Proceedings of ITP 2015*, volume 9236 of *LNCS*, pages 234–252. Springer, 2015.
13. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. 3540433767.
14. G. Nuka and J. Woodcock. Mechanising a Unifying Theory. In *Proceedings of UTP 2006*, volume 4010 of *LNCS*, pages 217–235. Springer, February 2006.
15. S. Obua. Partizan Games in Isabelle/HOLZF. In *Proceedings of ICTAC 2006*, volume 4281 of *LNCS*, pages 272–286. Springer, November 2006.
16. M. Oliveira, A. Cavalcanti, and J. Woodcock. A UTP semantics for *Circus*. *Formal Aspects of Computing*, 21(1):3–32, February 2007.
17. M. Oliveira, A. Cavalcanti, and J. Woodcock. Unifying theories in ProofPower-Z. *Formal Aspects of Computing*, 25(1):133–158, January 2013.
18. A. Pitts. Part III: The HOL Logic. In M. J. C. Gordon and T. F. Melham, editors, *Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic*, pages 191–232. Cambridge University Press, March 1993.
19. RTCA, Inc. Formal Methods Supplement to DO-178C and DO-278A. Technical Report DO-333, RTCA, Washington, DC 20036, USA, December 2011.
20. T. Santos, A. Cavalcanti, and A. Sampaio. Object-Orientation in the UTP. In *Proceedings of UTP 2006*, volume 4010 of *LNCS*, pages 18–37. Springer, 2006.
21. A. Sherif, A. Cavalcanti, H. Jifeng, and A. Sampaio. A process algebraic framework for specification and validation of real-time systems. *Formal Aspects of Computing*, 22(2):153–191, March 2010.
22. M. Spivey. The Consistency Theorem for Free Type Definitions in Z. *Formal Aspects of Computing*, 8(3):369–375, May 2015.
23. D. Traytel, A. Popescu, and J. C. Blanchette. Foundational, Compositional (Co)datatypes for Higher-Order Logic: Category Theory Applied to Theorem Proving. In *Proceedings of LICS 2012*, pages 596–605. IEEE, June 2012.
24. F. Zeyda and A. Cavalcanti. Higher-Order UTP for a Theory of Methods. In *Unifying Theories of Programming*, volume 7681 of *LNCS*, pages 204–223, 2012.
25. F. Zeyda and A. Cavalcanti. Mechanical reasoning about families of UTP theories. *Science of Computer Programming*, 77(4):444–479, April 2012.